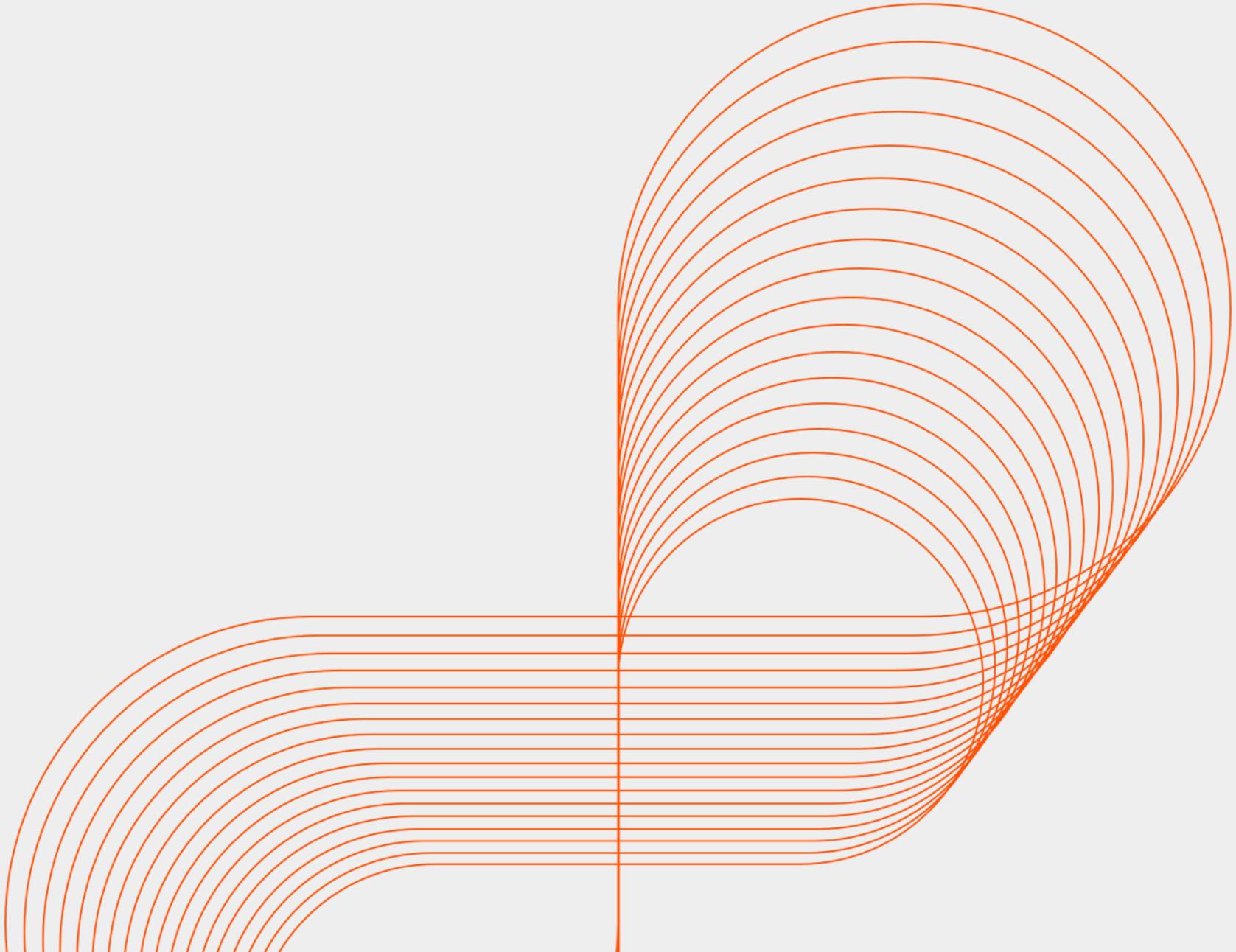
Persistent

Core Java: Logging API

Persistent University



Objectives :

- Logging and its importance
- Java Logging API java.util.logging
  - Logger, LogManager, Levels
- Managing Log messages
  - Exporting Log messages using Handlers
  - Formatting log messages using Formatters
- Managing Logs using Log4j API
  - Features of Log 4j
  - Understanding Appenders and Layouts
  - Logging in File
- Understanding slf4j

What is Logging?

- Process of recording and saving critical information about a system’s functioning
- Logging allows to report and persist error, warning and info messages so that these messages can later be retrieved and analyzed.
- Java contains the Java Logging API as java.util.logging package.
- The java.util.logging package provides the logging capabilities through Logger class.

Java Logging API

- Logger

A tool or API which is designed to let a program generate messages of interest to end users, system administrators, field engineers and developers.

- LogManager

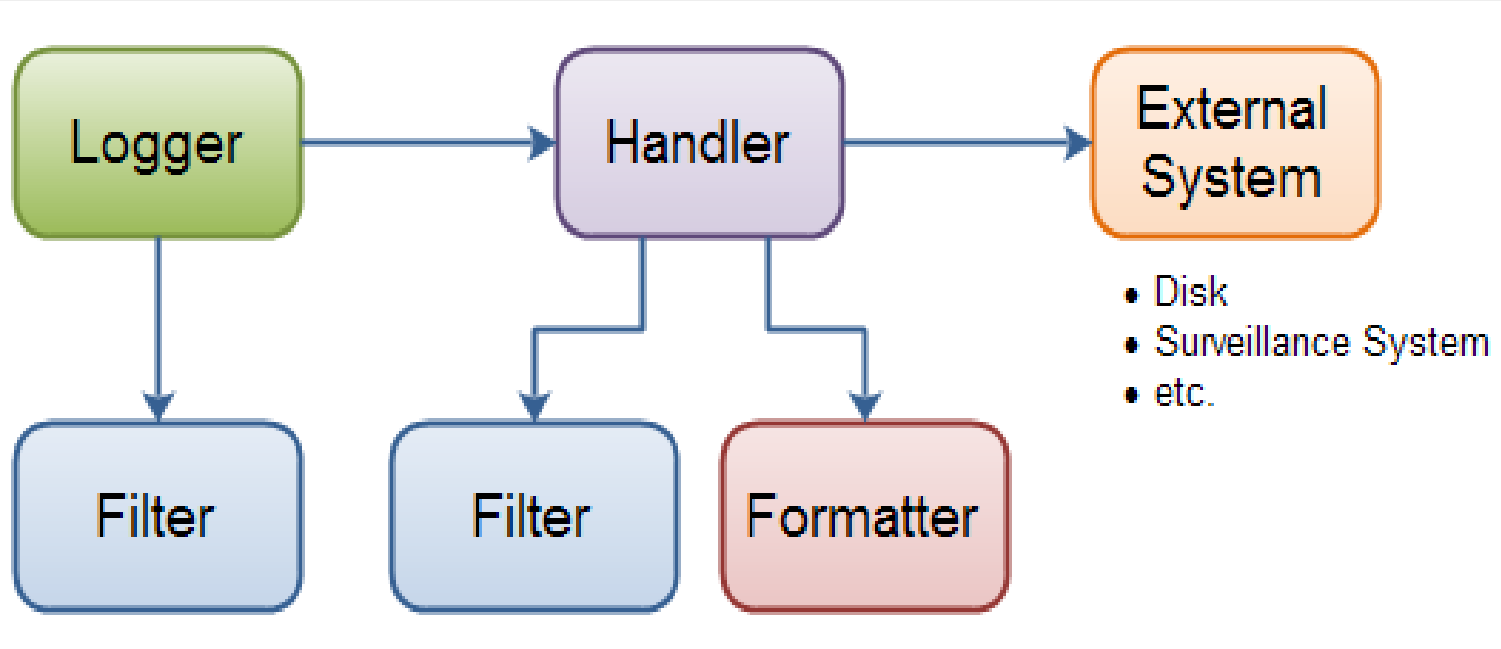
LogManager manages Logger which logs a LogRecord on a Handler at a specified Level.

- Handler

A Handler comes in varieties like ConsoleHandler, FileHandler, MemoryHandler and SocketHandler

Java Logging API overview

- How Java Logging API works?



Java Logging API overview

- All logging is done via a Logger instance which gathers the data to be logged into a LogRecord.
- The LogRecord is then forwarded to a Handler. The Handler determines what to do with the LogRecord.
- Logger's and Handler's can pass the LogRecord through a Filter which determines whether the LogRecord should be forwarded or not.
- A Formatter can be used by Handler to format the LogRecord as a string before it is sent to the external disk or system.

Logger

- This is how a Logger object is created
- Consider MyClass is the class to be logged.

```
private final static Logger LOGGER =
                    Logger.getLogger(MyClass.class.getName());
```

Log Levels

- The log levels define the severity of a message. The Level class is used to define which messages should be written to the log.
- Following are the log levels:
  - SEVERE (highest value)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (lowest value)

- In addition to this there are two levels as **OFF** and **ALL**. These will turn the logging off or to log everything.

Demo : Use of LogManager

```
import java.util.logging.*;
public class Sample {
        public static void main(String[] args) {
                LogDemo logDemo = new LogDemo();
                logDemo.divide(111, 0);
        } }
class LogDemo {
   static {
     LogManager.getLogManager().addLogger(
                Logger.getLogger(LogDemo.class.getName()));
   }
   private Logger logger =
        LogManager.getLogManager().getLogger(LogDemo.class.getName());

   public int divide(int x, int y) {
      int result = 0;
      try                          { result = x / y; }
      catch(ArithmeticException ex)   { logger.warning(ex.getMessage()); }
      return result;
   }
```

Log Handlers

- A Handler takes care of the actual logging to the outside world.
- One or more Handlers can be added to a Logger.
- When messages are logged via the Logger, the messages are eventually forwarded to the Handler's, if not rejected by a Filter or the minimum log level of the Logger.
- This is how to add handler.

```
logger.addHandler(new ConsoleHandler());
```

Types of Handlers

- Below are the types of Handlers:
  - ConsoleHandler : logs all messages to System.err
  - FileHandler : writes all messages to file.
  - StreamHandler : writes the log messages to an OutputStream.
  - SocketHandler : writes the log messages to some network address via a socket.
  - MemoryHandler : keeps the LogRecords internally in a buffer.

Demo : Using handlers

- Sample code

```
class LogDemo {
   static {
       LogManager.getLogManager().addLogger(
                       Logger.getLogger(LogDemo.class.getName()));     }
   private Logger logger =
LogManager.getLogManager().getLogger(LogDemo.class.getName());
   public LogDemo() {
                try {
          logger.addHandler(new FileHandler("my.log"));
          logger.setUseParentHandlers(false);
       } catch (SecurityException e) { logger.warning(e.getMessage()); }
        catch (IOException e)       { logger.warning(e.getMessage()); }
   }
   public int divide(int x, int y) {
       int result = 0;
       try{
                result = x / y; }
       catch(ArithmeticException ex)    { logger.warning(ex.getMessage()); }
       return result;
   }
}
```

Formatters

- Formatters are used to format the LogRecords before writing to an external system.
- There is an abstract class as Formatter. The two subclasses of Formatter which represent built in types are
  - **SimpleFormatter** : This formatter generates text messages with basic information. ConsoleHandler uses this formatter class to print log messages to console.
  - **XMLFormatter**: This formatter generates XML message for the log, FileHandler uses XMLFormatter as a default formatter.

Formatters

- A custom formatter can be created by extending Formatter class. This class will implement abstract format method. The String returned by the format() method is forwarded to the external system by the Handler.

Demo: Formatter

- This is an example of custom formatter.

```
public class MyFormatter extends Formatter {

@Override
public String format(LogRecord record) {
                return record.getLevel() + ":" +
record.getMessage();
 }
}
```

Demo

- Sample code

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.logging.ConsoleHandler;
import java.util.logging.FileHandler;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.LogManager;
import java.util.logging.Logger;

public class LoggingExample {
    static Logger logger =
Logger.getLogger(LoggingExample.class.getName());

    public static void main(String[] args) {
        try {
            LogManager.getLogManager().readConfiguration(new
                    FileInputStream("mylogging.properties"));
        } catch (SecurityException | IOException e1) {
            e1.printStackTrace();
}
```

Demo

- Sample code

```
  logger.setLevel(Level.FINE);
  logger.addHandler(new ConsoleHandler());

//adding custom handler
   logger.addHandler(new MyHandler());
   try {
          Handler fileHandler = new FileHandler(“C:/logger.log", 2000, 5);
           fileHandler.setFormatter(new MyFormatter());
           //setting custom filter for FileHandler
           fileHandler.setFilter(new MyFilter());
           logger.addHandler(fileHandler);
           for(int i=0; i<1000; i++){
                 logger.log(Level.INFO, "Msg"+i); }//logging messages
           logger.log(Level.CONFIG, "Config data");

      } catch (SecurityException | IOException e) {
           e.printStackTrace();
       }   }
  }
```

log4j

- log4j is a reliable, fast and flexible logging framework (APIs)
- It is written in Java and distributed under the Apache Software License.
- It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog, etc.

Components of log4j

- log4j has three main components:

  - **loggers**: captures the logging information.
  - **appenders**: publishes the logging information to various preferred destinations.
  - **layouts**: formats the logging information in different styles.

Features of log4j

- Thread-safe.
- Optimized for speed.
- Based on a named logger hierarchy.
- Supports multiple output appenders per logger.
- Supports internationalization.
- Not restricted to a predefined set of facilities.
- Logging behavior can be set at runtime using a configuration file.
- Designed to handle Java Exceptions from the start.

Log levels for log4j

- The log levels are as
  - ALL
  - TRACE
  - DEBUG
  - INFO
  - WARN
  - ERROR
  - FATAL.

Log4j API

- log4j API follows a layered architecture. Through this each layer provides different objects to perform different tasks.
- This layered architecture makes the design flexible and easy to extend in future.
- The two types of objects available are :
  - **Core Objects:** These are mandatory objects of the framework. They are required to use the framework.
  - **Support Objects:** These are optional objects of the framework. They support core objects to perform additional but important tasks.

Core objects

- Core objects are
  - **Logger object :** responsible for capturing logging information and they are stored in a namespace hierarchy.
  - **Layout object :** used to format logging information in different styles. Also it supports the appender objects before publishing logging information. Layout objects publish the logging information in human-readable and reusable way.
  - **Appender object :** responsible for publishing logging information to various preferred destinations such as a database, file, console, UNIX Syslog, etc.

Support objects

- The support objects are
  - **Level object :** defines the granularity and priority of any logging information.
  - **Filter object :** used to analyze logging information and make further decisions on whether that information should be logged or not.
  - **ObjectRenderer :** specialized in providing a String representation of different objects passed to the logging framework.
  - **LogManager :** manages the logging framework.

Log4j configuration

- The **log4j.properties** file is a log4j configuration file which keeps properties in key-value pairs. By default, the LogManager looks for a file named **log4j.properties** in the **CLASSPATH**.
- The syntax for log4j.properties file is as follows.

```
# Define the root logger with appender X log4j.rootLogger =
DEBUG, X # Set the appender named X to be a File
appender log4j.appender.X=org.apache.log4j.FileAppender #
Define the layout for X appender
log4j.appender.X.layout=org.apache.log4j.PatternLayout
log4j.appender.X.layout.conversionPattern=%m%n
```

Log4j configuration

- Sample code

```
# Define the root logger with appender file
log4j.rootLogger = DEBUG, FILE

# Define the file appender
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${log}/log.out

# Define the layout for file appender
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Log4j configuration

- The level of the root logger is defined as **DEBUG**, The **DEBUG** the appender named **FILE** to it.
- The appender **FILE** is defined as **org.apache.log4j.FileAppender**. It writes to a file named **log.out** located in the **log** directory.
- The layout pattern defined is *%m%n*, which means the printed logging message will be followed by a newline character.

Appenders

- Appender objects are responsible for printing logging messages to different destinations such as consoles, files, sockets, NT event logs, etc.
- Various properties are as

Property	Description
layout	Appender uses the Layout objects and the conversion pattern associated to format the logging information.
target	The target may be a console, a file, or another item depending on the appender.
level	The level is required to control the filtration of the log messages.
threshold	Appender can have a threshold level associated with it independent of the logger level. It ignores any logging messages of lower level than the threshold level.
filter	The Filter objects can analyze logging information beyond level matching and decide whether logging requests should be handled by a particular Appender or ignored.

Layout

- Layout provides support to appender objects before publishing logging information.
- Various layouts available are
  - DateLayout
  - HTMLLayout
  - PatternLayout
  - SimpleLayout
  - XMLLayout

Demo : log4j in Java application

- Sample code

```
import java.io.*;
import java.sql.SQLException;
import java.util.*;
public class log4jExample{
/* Get actual class name to be printed on */
static Logger log =
                Logger.getLogger(log4jExample.class.getName
());
                public static void main(String[] args)throws
                IOException,SQLException{
                                log.debug("Hello this is a debug
message");                                log.info("Hello this
is an info message");
                 }
 }
```

Logging methods

Method	Description
public void debug(Object message)	It prints messages with the level Level.DEBUG.
public void error(Object message)	It prints messages with the level Level.ERROR.
public void fatal(Object message)	It prints messages with the level Level.FATAL.
public void info(Object message)	It prints messages with the level Level.INFO.
public void warn(Object message)	It prints messages with the level Level.WARN.
public void trace(Object message)	It prints messages with the level Level.TRACE.

Slf4j - Simple Logging Facade for Java

- The Simple Logging Facade for Java.
- It serves as a simple facade or abstraction for various logging frameworks, such as java.util.logging, logback and log4j.
- It allows the end-user to plug in the desired logging framework at the time of deployment.
- It supports various logging frameworks.

Slf4j : required configuration

- SLF4J-enabling requires the addition of a single dependency, i.e. *slf4j-api-1.7.21.jar*.
- **slf4j-log4j12-1.7.21.jar :** Binding for log4j version 1.2, a widely used logging framework. You also need to place log4j.jar on your class path.

- **slf4j-jdk14-1.7.21.jar :** Binding for java.util.logging, also referred to as JDK 1.4 logging

Slf4j : required configuration

- **slf4j-nop-1.7.21.jar :** Binding for NOP, silently discarding all logging.
- **slf4j-simple-1.7.21.jar :** Binding for Simple implementation, which outputs all events to System.err. Only messages of level INFO and higher are printed. This binding may be useful in the context of small applications.

Basic rules of slf4j

- We can switch from one logging framework to another by replacing slf4j bindings on class path.
- SLF4J does not rely on any special class loader machinery.
- Each SLF4J binding is hardwired at compile time to use only one specific logging framework.

Slf4j Vs log4j

- SLF4 is a logging facade Java.
- It is not a logging component and it does not do the actual logging.
- It is only an abstraction layer to an underlying logging component.

- **Slf4j :**

- It is a logging component and it does the logging instructed to do.

- **Log4j :**

Limitations of logging

- It can slow down an application.

- If too verbose, it can cause scrolling blindness.

Summary :

- With this we have come to an end of our session, where we discussed about ….
  - The concept of logging
  - Why logging is required
  - Various ways to perform logging as logging API, log4j, slf4j
  - How to implement applications for logging.

- At the end of this session, we see that you are now able to answer following questions:
  - Develop applications with various logging APIs

Appendix

References

Thank you

Reference Material : Websites & Blogs

- http://www.vogella.com/tutorials/Logging/article.html
- http://tutorials.jenkov.com/java-logging/overview.html
- http://www.tutorialspoint.com/log4j/index.htm
- http://www.slf4j.org/manual.html

Persistent

Thank you!

Persistent University